Word Tiles, Building a Scrabble-like Word Game with Elixir, Phoenix, and React

Melisa Brown, Chuhan Zhang

October 25, 2019

Introduction and Game Description

We built a scrabble-like board game named "Word Tiles" that can be played with multiple players, using React, Phoenix, and Elixir. Upon entering the game's main page, users will be asked to enter their names and have the option of joining an already occupied game room or starting their own game room. Once a game has started, additional players cannot join. Within a game room, players in the game have access to a game chat that is unique for the game.

The object of the game is to accumulate the most points by forming words on a 15 x 15 board, where a slot corresponds to a letter. In total there are 100 letter tiles available for play. Each letter has a point value and distribution that roughly corresponds to its frequency in a given language. The less frequently a letter occurs in the language, the higher the point value.

The game starts by drawing 7 tiles (without replacement) from the "bag" of 100 tiles for each of the players. The turn order will be determined by when the player joined the game. Players take turns forming words by placing the tiles on the board. A turn consists of playing a word or passing. As long as there are still tiles left in the "bag", the game will automatically replenish the number of tiles played in a turn, so that players will always have 7 tiles available for play. If there are no tiles left in the bag, the game will continue with whatever tiles are in play. The game ends when no tiles are left in the "bag" and A) a player has used all of his/her letters or B) no valid word can be formed by any of the players. The player with the most points at the end of the game wins.

Rules for tile placement. At the start of the game, the first player must form a word that uses the center position on the board indicated by a star. Words must be at least two letters long and be oriented left to right or top to bottom. Words cannot be formed diagonally, right to left, or bottom to top. All letters in the word must fit on the board and use at least one letter tile that is already in play.

Scoring. The score for a turn is the sum of the points for letters in the word played. This includes letters that were already on the board and any additional letters played. The game board has bonus positions labeled "DL", "DW", "TL", and "TW" corresponding to double letter, double word, triple

letter, and triple word. Placing a tile on any of these positions will adjust the word score accordingly. For example, playing the word "CAT" on normal positions totals: C =¿ 2 + A =¿ 1 + T =¿ 1 = 4 points. If the position of the letter "C" had a "TL" the word would be worth (2 * 3) + 1 + 1 = 8 points. If the position had a "TW", the word would be worth (2 + 1 + 1) * 3 = 12 points. Additionally, players get a 50 point bonus for playing all seven tiles in a turn.

UI Design

The first page that the player encounters is a landing page where they are asked to enter their name and game name. This is implemented in word_tiles_web/ templates/page/game.html.eex. Once they submit their information, they will join a default Phoenix channel. The GameSupervisor implemented in lib/ word_tiles/ game_ supervisor is then responsible for starting GameServer processes dynamically. The GameServer uses GenServer to store the state of a given game.

We used a react app for displaying the game itself. These files are contained in the assets/js folder:

wordtiles.jsx. The main stateful React component that renders the game state using the react-konva library. It contains functions to handle tile moves, update tiles when the player clicks the submit or pass button.

board.jsx. A stateless React component that renders the board using the react-konva library. It contains functions for creating the board grid, rendering bonus tiles which need to be labeled as such, and rendering played tiles on the board.

playertiles.jsx. A stateless React component that renders the tiles in play for each player using the react-konva library. The tiles can be individually dragged onto the board and their positions are recorded.

UI to Server Protocol

The first message that gets sent from the browser are the sockets params that are passed from the client and can be used to verify and authenticate a user. In our case there is no user token, so all users are verified. After users are verified they are assigned to a socket. This allows us to reference the player in the callback function of the channel module. This is handled in lib/word_tiles_web/channels/user_socket.ex.

The lib/word_tiles_web/channels/games_channel.ex file contains game-specific messages that flow over the channel. The first function is the join callback function which assigns players to a game/topic once they are verified. Two handle_in callback functions are implemented. One to handle words that are submitted for play and the other to handle chat messages.

Data Structures on Server

The main data structure on the server is a map that holds the game state. The map consists of seven key-value pairs, where keys are the atoms: board, bag, letters_left, players, dictionary, winner, and current_player.

> **board**: holds a list of 225 map items corresponding to a slot in the 15x15 board. Each slot map has a position, letter and bonus key. "Position" is an integer from 0 to 224. "Letter" is by default set to blank and is updated if a player placed a letter in the slot when playing a word. "Bonus" indicates whether the slot has a bonus scoring opportunity. The "bonus" field is blank for most slots, but contains a two letter abbreviation "DL", "DW", "TL", or "TL" for slots that have double letter, double word, triple letter, and triple word scoring potential, respectively. There are 8 such slots, but we intended to have more.

> **bag**: contains a similar data structure to board, a list of maps. Each map represents a letter tile which has the keys "letter, "qty", and "points". There are 26 map items in the list for each of the letter in the English language. Unlike the original Scrabble game, our game does not have a blank tile option. Each letter has a specific score and starting quantity. The bag data structure simulates a physical bag having 100 tiles. As players submit words, their letters are replenished from the bag until there are no letter left. "Letters_left" is an integer containing the number of letters left in the bag. This is viewable by the players and decremented as tiles are drawn from the bag. "Players" contains a list of player maps. Each player map has "name", "letters", "score", and "turn" keys. Name has a the player's name as a string. Letters contains a list of letter tiles available to play for each player. Score and turn contain integer values.

> **dictionary**: has a value consisting of a list of words. We used a text file of 180,000 words taken found here: https://scrabutility.com/TWL06.txt . The consequence of this is that the entire list is read for each game. Ideally we would have liked to keep the list of words separate from the game data structure.

> **winner** is nil until there is a winner for the game. When to game is over, "winner" will contain the name of the winning player as a string. "Current_player" contains a string of the current player's name and is updated when a player submits a word or passes.

The other data structure that is returned by the server is the client view. This is a map that contains the board as described above, letters_left, and player_tiles. "Player_tiles" is a list of letters that is generated per player.

Implementation of game rules

Game rules were mostly implemented via the server-side logic. We did not fully implement our intended game rules, and had an especially difficult time figuring out how to connect the client and server-side. On the server-side, a player must be listed as the current player in the game state in order to play a word on the board. We did not successfully implement verifying that the word played is in the dictionary or verifying valid tile placement.

Challenges and Solutions

Multiplayer server side logic. Initially our intention was to allow the game to be played with a varying number of players, at least two and at most four players. We also wanted users to be able to select which game to join from a list of available games. This would have necessitated having an additional field when the game was created that indicated the number of players that could join the game. We would also need a way to list all games currently being played. As implemented, the number of players cannot be set. Another challenge with implementing a multiplayer game was figuring out a way for the server to sync the states of multiple players. We resolved this but having one game state that was continually updated as players made moves. The server also need to ignore or prevent moves coming from other players (browsers) when one player has current turn. We enforced the turn-based rules on different players by keeping track of the current player and implementing a check at the beginning of any moves submitted to the server. Also once targeted number of players have joined the table, server need to prevent more players from joining the game. We do not currently have an enforced limit on the number of players that can join the game.

Chat room server side implementation. As stated in the project instructions, an unlimited number of people should able to observe the game and need to communicate with each other by text chat. We predicted that implementing the observer logic that can only let them type text messages without giving them permission to change the current game state would be challenging. We were not able to implement the chat server as described.

Check the validity of played word. Checking of validity need to be implemented on the server side. So we need to obtain a dictionary of words in English(later on other languages such as French, Spanish) or delegating word checking to other online dictionary sources. We did find a dictionary of 180,000 English words and successfully associated with the game. We wrote a simple function that checks if a word is listed in the dictionary and returns a boolean value. However, we did not finish writing a function that could determine what word was played from the current tiles on the board and the new tiles that the player submitted. Additionally, we would have liked to isolated reading the dictionary file into a completely separate module and function that could be called from any game. It is inefficient to hold such a long list of words in the state of each game.

Connecting the client-side application to the server-side logic via channels. We determined that we would need three general actions that would have to be included in the game channel: submitting a word to play on the board, submitting a chat message, and notifying the players of an update game state. What was unclear was the number and type of callback functions needed. We did not successfully implement the chat functionally but surmised that it could be done with a single handle_in callback function to get the player's message and broadcast it on the channel.